



PGR112 – Step 3: Classes and Objects

Object oriented programming

Bogdan Marculescu / bogdan.marculescu@kristiania.no

Agenda

- Classes and objects
- Constructors
- Overloading
- Fields
- Getters and setters
- Access modifiers
- this
- enum

Classes

- The class is the starting point for objects we want to create. We describe in our class:
 - What attributes (fields) the objects have
 - What methods do the objects have (ie what can the objects do)
- When we create (or initialize) an object of a class, we use a special method called a constructor, and we call it in a special way: new
- Common to all objects from the same class is that they share methods and attributes. But objects can be different because they have different values on the attributes. The objects then have different state (state).

Objects

- The objects are thus "born" when we create them via new and they are created on the basis of a selected class.
- When we create them, we usually want to be able to do something with them. We need a reference to them. Let's take a (modified) example from [Oracle](#).
- Read more about classes and objects [here](#).

Bicycle example

`Bicycle bicycle = new Bicycle(); ->`

I am creating a new object of the type Bicycle.

The reference to the object is *bicycle*.

`bicycle.printStates(); ->`

I am calling the object's method "printStates()", using the reference to make it clear I mean the states of the instance I have created.

printStats()

- We see that the printStates () method prints information about the state of the object by printing information about the values of the fields in the object:

```
void printStates() {  
    System.out.println("Speed:" +  
        speed + " gear:" + gear);  
}
```

- There are more elegant ways to do it, but so far we can rest assured that strings can be joined by using + between them.

Constructor

- But where is the constructor we used? I can not find it in the Bicycle class?
- All classes have a standard constructor (although we do not make it ourselves). We used this standard constructor when we made a bicycle object. The standard constructor has no parameters.
- Constructors look like this:
 - The name matches the name of the class (Ex: Bicycle)
 - It can have parameters (it is a method)
 - It returns the object.
 - It is usually public
- The constructor is a special method of creating objects. If nothing special is going to happen when creating an object, then we do not need to create it (as we just saw).
- The constructor is most often used to set start values for the attributes of the object.
- Let's try to use a constructor in the example...

So, what we learned

- When we make a constructor, we no longer have a standard constructor available.
 - When I made a constructor that took two parameters (speed and gear), I no longer have an empty constructor available. I MUST send with both speed and gear to be able to create an object of type Bicycle.
- But we can make more constructors!
- Let's create an empty constructor as well (one that does not take parameters).

Method overloading

- We can therefore have several constructors, as long as they take different parameters. This also applies to common methods in a class, and is called method overloading: Methods can have the same name in a class, as long as they take different parameters.

For example:

```
public void print(String s){}
```

```
public void print(int i){}
```

```
public void print(int i, String s){}
```

```
public String print(String s){}
```

Constructor overview

- Constructor: a special method that we use in connection with the creation of an object.
- We call it using new, and send with any initial parameter values.
- We can have several constructors in the same class (as long as they have different parameters).
- The constructor ensures that the object is created with the correct initial values.

What about String?

- We have already worked with Strings. And we then worked with objects. But what about the constructor?
- We have made a String like this:
- `String s = "Anything between two double quotes";`
- But we can choose to make the same string like this via constructor:
- `String s = new String ("Anything between two double knots");`
- And the String class has several constructors we can choose from.
- This is not relevant right now, but know that the String class is a bit special. Those of you who want to read more about this can take a look [here](#).

Attributes and objects

- The attributes in the objects are often called fields, instance variables or member variables.
- These are variables that we have access to in the methods in the object.
- We saw that in our example. The printStates method printed the value of its fields. The method had access to the object's variables.
- We can choose whether fields (and methods) can also be accessed from outside the object using access modifiers.

Access modifiers

- We have previously seen that we have had public methods (public static void main...). Slightly simplified, we can say that with the public we open up for something to be accessed from outside the class / object. We do not go into detail on this yet, but there will be more about this when we start with inheritance later in the course.
- What we are going to take a look at now are some access modifiers for fields and methods. In today's example (Bicycle) we saw little of them. No access modifiers mean they have default access.
- The table below shows the 4 options we have, and we will initially focus on private and public.

Only accessible in the class

Accessible «everywhere»

Access Modifier	within class	within package	outside package by subclass only	outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

javatpoint.com

Private

- We want to limit access to fields and methods (above other objects). We should make sure that we only give access to what is needed.
- It is common to set fields to be *private*. Then we make sure that only the object itself can change its state. We give the object better control over its own state.
- Private methods are often called auxiliary methods. There are methods that are not offered to anyone other than the object itself. It can, for example, be a complicated (*public*) method that is divided into several smaller help methods (*private*).

Public

- With public, we provide explicit access. A public field means that others than the object itself can change the value.
- It is common to combine private fields with public getters and setters.
- Guessing: A method of extracting a value of a field in an object.
- Sets: A method of changing a value of a field in an object.
- Let me demonstrate in today's example...

IntelliJ and getters/setters

- It is so common to make getters and setters that IntelliJ has the support to make them in a simple way.
- See demonstration

Encapsulation

- To quote [tutorialspoint](#):

«**Encapsulation** is one of the four fundamental OOP concepts. The other three are inheritance, polymorphism, and abstraction.

Encapsulation in Java is a mechanism of wrapping the data (variables) and code acting on the data (methods) together as a single unit. In encapsulation, the variables of a class will be hidden from other classes, and can be accessed only through the methods of their current class. Therefore, it is also known as **data hiding**.

To achieve encapsulation in Java –

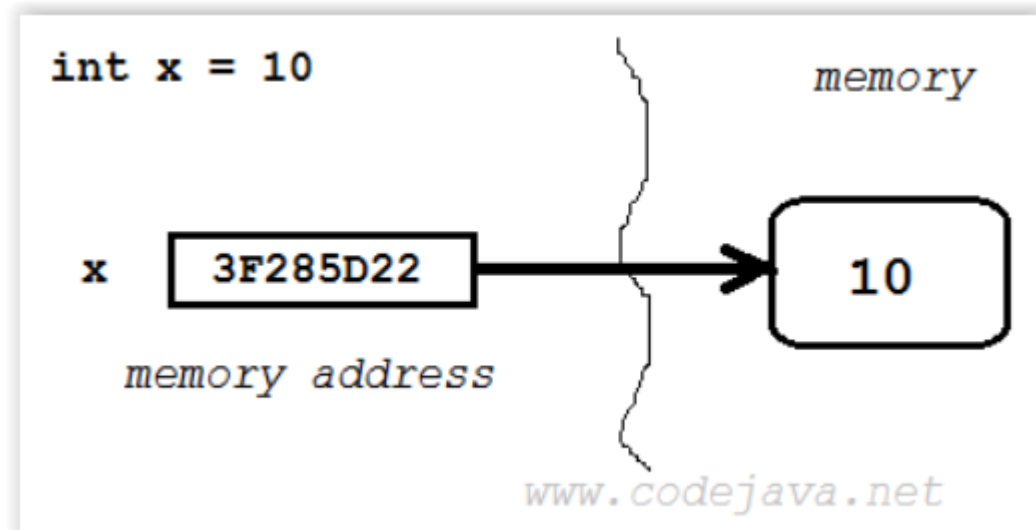
- Declare the variables of a class as private.
- Provide public setter and getter methods to modify and view the variables values.»

Quick recap

- In step 2, we worked (mainly) with primitive data types inside methods (local variables).
- In this step (step 3) we have learned to create objects, and we have seen how we can have variables in an object (fields).

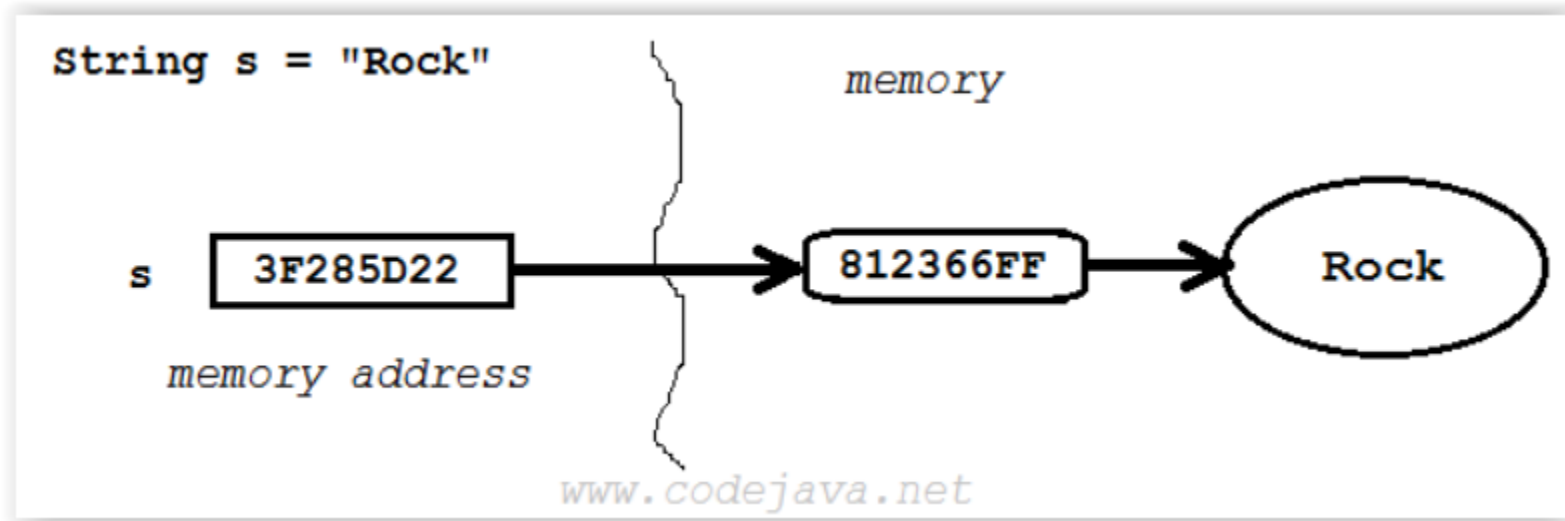
Variables

- Primitive variables (int, char, boolean, etc.)

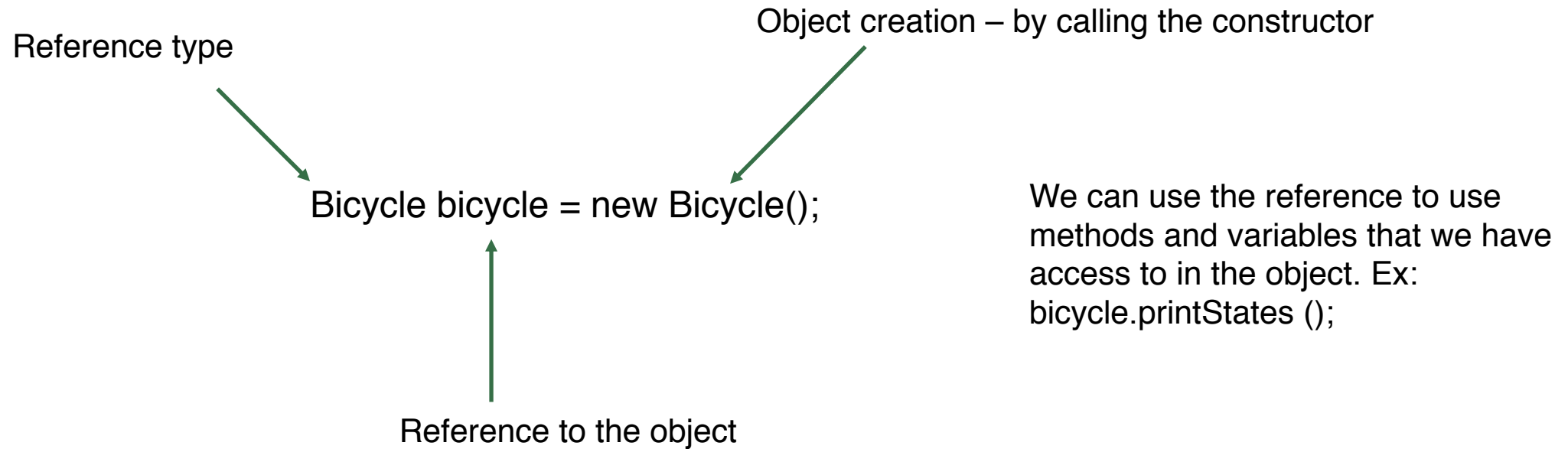


Reference variable

- Reference variables – references to an object



Today's example was a reference variable



Fields - summary

- Fields are variables that apply to an object. They constitute the state of the object (state).
- Fields will have a start value when the constructor has done his job of creating the object.
- We can refrain from setting values via the constructor. They will then have a default value.
- We want to hide fields from the outside world. We do this by making them private. Then only the object itself can access them.
- But we can offer both read and change rights anyway. We do this by offering methods (getters and setters).

Getter

- A method that returns the value of a field in the object. For example:

```
private String name; //Field
```

```
public String getName() { return name;}
```

Setter

- A method that changes the value of a field in the object. For example:

```
private String name; //Field
```

```
public void setName(String newName) { name = newName;}
```


this

- But what if the name of the parameter in the setter method is the same as a field in the object? For example:

```
private String name; //Field
```

```
public void setName(String name) { what do we do now? } // Setter method
```

The field and the parameter have the same name



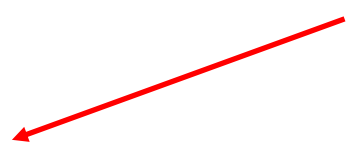
- In this case, the object must have a reference to itself:

```
private String name;
```

```
public void setName(String name) { this.name = name; }
```

This object's name is set equal to the value of the name we receive in the setter method.

The use of «this» is also common in constructors.



Enum

- We remember enum as a data type in DB. We also have Enum available as a type in Java.
- We can make enums inside a class, or outside.
- The values we define in an enum we write in UPPERCASE LETTERS, as they are constants.
- We can make quite advanced enums with constructors and methods, but in the first instance we stick to simple enums that only hold a fixed set of constants.

Example (from [geeksforgeeks](https://www.geeksforgeeks.org/enums-in-java/)):

```
// A simple enum example where enum is declared
// outside any class (Note enum keyword instead of
// class keyword)
enum Color
{
    RED, GREEN, BLUE;
}
```

Enum use

- When we want to use a value in our enum, we can refer to it like this:
- `Color c = Color.RED;`

Wrap-up

- The goals of this step are:
 - I can create classes with fields and methods.
 - I can make constructors, and I understand how they work.
 - I can create objects.
 - I can use a reference variable to access an object (and I understand what that means).
 - I can make getters and setters and I understand the purpose of this (encapsulation).
 - I can make an enum and I can use it in a field.
 - I know that an object can use this to refer to itself.
- Good luck with the tasks!
- Remember that you have help available all week 😊